n 番目に小さい値を求めるアルゴリズムについて

倪　　永茂

Algorithms for Finding nth Smallest Element in Unsorted Array

NI Yongmao

『宇都宮大学国際学部研究論集』（ISSN1342-0364）第 48 号（2019 年 9 月）抜刷

JOURNAL OF THE SCHOOL OF INTERNATIONAL STUDIES
UTSUNOMIYA UNIVERSITY, No.48 (September 2019)

n 番目に小さい値を求めるアルゴリズムについて

倪　　永茂

はじめに

膨大なデータは活用するには、定められた規則に沿って順序づけることがよく行われる。順序づけのことは専門用語では整列（ソート、Sort）という。順序はひとつの基準によって決めるケースもあれば、ひとつの基準で定まらない場合は、さらに第2、第3の基準を増やし、一意に決めるようにすることが多い。

たとえば、不動産屋さんが賃貸物件をリストアップする際に、顧客のニーズを想定し、家賃を第一の整列基準にし、同一家賃の物件に対して、さらに面積の広さや、築年数、最寄り駅までの徒歩所要時間等を、第2、第3、第4の基準にしてリストをつくったりする。順序づけの規則はその都度変更されることはよくあるが、規則が決まれば物件データの順位が一意に決められたほうが一般的に望ましいとされる。

本文は与えられた順序づけに関する規則のもとで、その規則に沿って *n* 番目のデータを求めることを論ずるものである。本文のタイトルにある「*n* 番目に小さい値」というのはあくまでもわかりやすさを優先した日本語の表現であり、本質的には、規則に沿った *n* 番目のデータということを意味する。

なお、規則に沿って決められた順序の最初を日本語では一般的に1番目（英語 first）というが、0番目という表現も近年よく見られるようになったので、注意して両者を区別しなければいけない。本文では、混乱を避けるために、順序の最初を1番目と呼ぶことにする。

また、*n* 番目の意味はひとにとって使われる文脈によって異なるが、本文でいう *n* 番目とは *n*-1 番目までのデータを取り除いたら1番目となるデータの値のことをいう。例示すれば、5, 2, 0, 2, 6, 10 というデータに対して、1番目は0、2番目は2、3番目は2、4番目は5、5番目は6、6番目は10ということを意味する。値が2のデータが重複しているので、2番目の2と3番目の2はそれぞれ、どの2を指すのかは、基準ひとつだけでは決まらない。

本文の内容に密に関係するものとして、最小値、最大値、中央値等があげられる。データが小さい順に整列され、その数が奇数の N 個だとすると、最小値、最大値はそれぞれ $n = 1$、$n = N$ 番目のデータに対応し、中央値は $n = (N+1) / 2$ 番目のデータに対応するものとすることができるので、最小値、最大値、中央値の求め方が「*n* 番目に小さい値を求めるアルゴリズム」に含まれる。

n 番目に小さいデータを求める最善のアルゴリズムは与えられた状況(条件) によって異なる。本文では、順序づけに関連するさまざまなアルゴリズムについて考察していき、最善のアルゴリズムを提示する。

なお、自明なことであるが、小さい順で整列済のデータ配列において、*n* 番目に小さい値は先頭から *n* 番目のデータであるので、計算時間量は $O(1)$ である。ただし、線形リストのようなデータ構造では、*n* 番目をアクセスするのにリンクをたどるので、$O(n)$ になる。データ構造が計算時間量を大きく左右することがある。

Ⅰ 適切な比較キーの作成

データを順序づける際に、比較対象となる項目（フィールドともいう）をここでは比較キーと呼ぶことにする。比較キーは整列では、ソートキーとも呼ばれる。比較対象が複数にまたがる場合に、比較キーは第1基準となる項目、第2基準となる項目、第 k 基準となる項目を合わせたものになる。

比較回数がアルゴリズムの計算時間量を大きく左右するので、複数の項目をひとつの比較キーにまとめることができれば、計算時間がより短縮される。

以下では、複数の項目を異なるビットにエンコードする手法で、ひとつの比較キーにまとめる方法を

紹介する。ただし、本方法が成立する前提は各項目の最大値が前もってわかることである。といっても、各項目の最大値および最小値を前もって見積もることが、プログラムの設計や開発の基本なので、実用に供するプログラムはほとんどがこの前提を満たしているといえよう。

項目の数を k、複数の項目をそれぞれ a_i、最大値をそれぞれ $2^{mi} - 1$ 以下とすると、比較キーは $\sum_{i=1}^{k} (a_i \ll M_i)$ とする。ただし、「<<」は C 言語のビットシフト演算子を表し、$M_i = \sum_{j=i+1}^{k} m_j$ とする。

実例として、スポーツ選手間の順序づけのために第 1 基準を体重 w（最大値 120 kg）、第 2 基準を身長 h（最大値 250 cm）として比較キーを作成してみる。$k = 2$、$a_1 = w$、$m_1 = 7$、$a_2 = h$、$m_2 = 8$ とすることにすれば、まとめられた比較キーは $(w \ll 8) + h$ となる。体重 w が異なれば、身長 h の値の如何にかかわらず、順序が決まる。また、w が同じ（同一体重）のスポーツ選手については、身長 h の値の違いによって順序が決まる。

体重 w は昇順（値が小さいほうが順序的に前）、身長 h が逆に降順（値が大きいほうが順序的に前）という、順序的に相反する複数の項目をひとつの比較キーにまとめることも可能である。その項目の最大値から引いた値を使えば良いからである。上記の例では、比較キーが (w << 8) + (256 − h) になる。

このように、複数の項目による順序づけは単一比較キーのそれに帰着することが可能なケースが多いので、簡単化するため、以下の議論では単一キーの比較に限定することにする。

なお、比較キーを工夫すれば、n 番目に大きい値を求めることは n 番目に小さい値を求めることと同価であることは上記の議論で明らかであろう。

Ⅱ 各アルゴリズムに対する考察

1 Excel の Small 関数、Large 関数

Microsoft 社の表計算ソフト Excel には、n 番目小さい（または大きい）関数として、SMALL、LARGE が提供されている。

使い方は共通していて、= SMALL（配列 , 順位）という形になる。配列には比較キーであるデータ配列、順位には求めたい n 番目の数字を入れればよい。データ配列を整列しておく必要はない。

最小値を求める MIN 関数は、SMALL 関数において順位 $n = 1$ とした時に算出結果と同じである。

2 選択アルゴリズム

選択アルゴリズム(selection algorithm) とは、データの配列から n 番目に小さい値を探すアルゴリズムであり、本文の議論対象そのものである。

良く知られている選択アルゴリズムのひとつは以下である。すなわち、データ配列をまず、5 つずつの要素の小配列に分割する。次にその小配列ごとに中央値を探し、その各中央値だけを抽出した配列に対して、さらに選択アルゴリズムを再帰的に適用する。そうして見つかった中央値の中央値は、整列アルゴリズムとして知られているクイックソートのピボット値として最適であり、1 回の再帰計算で、配列の個数を一定の割合で減らすことができる。その結果、最悪でも計算時間量が $O(n)$（n はデータの個数、すなわち、データ配列の長さ）の選択アルゴリズムが得られる。しかしこのアルゴリズムは実装が複雑であるため，実用上はクイックソートに類似した以下のアルゴリズムのほう（計算時間量が同じく $O(n)$ ）が高速だといわれている。

```
// n 番目に小さい値を算出する
int selection(int *a, int sz, int n) {
  int i, j, p, l, r, t;
  l = 0, r = sz-1, n--;
  while (l < r) {
    p = a[(l+r)/2];
    i = l-1, j = r+1;
    while (1) {
      while (a[++i] < p);
      while (a[--j] > p);
      if (i > j) break;
      t = a[i], a[i] = a[j], a[j] = t;
    }
    if (n < i) r = j;
    if (n > j) l = i;
  }
  return a[n];
}
```

上記の C 言語による関数の記述では、引数はそれぞれ、データ配列 a[]、データ配列の個数 sz、求めたい順位 n を意味する。データ配列 a[]={10,5,0,2,2,6} に対して、4 番目に小さい値

を求める selection(a, sz, 4) を呼び出せば、関数の戻り値 5 が得られる。

3 バイナリサーチ（2 分探索）関連

バイナリサーチ（binary search）は整列済のデータ配列に対する探索アルゴリズムの一つである。

探索にあたって、中央の値を見て、探索したい値（探索キー）との大小関係を調べ、探索キーが中央の値の前（あるいは左）にあるか、後（あるいは右）にあるかをみて、探索の範囲を片側（データ数が半分）に縮小していく。その結果、バイナリサーチ・アルゴリズムは時間計算量が $O(\log n)$ である。ただし、n はデータの個数である。

バイナリサーチは探索キーがデータ配列に存在するかどうかを調べることに対し、以下の 2 つの関数はより実用的であろう。

lower_bound（）： データ配列 a のインデックスのうち、探索キー以上となる最小のインデックスを求める。つまり、a[index]>=key という条件を満たす最小の index。

upper_bound（）：データ配列 a のインデックスのうち、探索キーより大となる最小のインデックスを求める。つまり、a[index]>key という条件を満たす最小の index 。

両関数の差がわずかだと思われるかもしれないが、実際には用途が大きく異なる。

例をみてみよう。整列済データ配列は [0,2,2,5,6,10] とする。C 言語では配列のインデックスが 0 からスタートという点に留意すると、各探査キーに対し、それぞれの関数が求めたインデックスが表 1 になる。

ただし、探索キー -1、および 11 については、データ値の範囲外にあるので、得られた結果は関数の実装によって異なり、注意を要する。

lower_bound 関数と upper_bound 関数を組み合わせて使うこともよくある。例えば、値が 2 のデータの個数を知るには、upper_bound（2）-lowe_bound（2）のようにすればよい。値が 2 のデータの個数が log（n）以下の場合は、lower_bound（2）で得られたインデックスからひとつずつ（線形的に）確認していくほうが高速だが、それ以上の個数の場合は、前者のほうがより高速である。つまり、全体のデータ個数が 2^m 程度であり、探索キーの個数が m 以上であれば、前者を利用すべきであろう。

表 1 データ配列 [0.2.5.6.10] に対する index

探索キー	lower bound	upper bound
-1	0	0
0	0	1
1	1	1
2	1	3
3	3	3
4	3	3
5	3	4
6	4	5
7	5	5
8	5	5
9	5	5
10	5	6
11	6	6

C 言語で記述する lower_bound 関数、upper_bound 関数はそれぞれつぎに示す。

```
// a[i]>=key という条件を満たす最小の i
int lower_bound(int *a, int sz, int x) {
  int m, l = 0, r = sz;
  while (l+1 < r) {
    m = (l+r)/2;
    if (a[m] < x) l = m; else r = m;
  }
  return a[l] < x? r: l;
}
// a[i]>key という条件を満たす最小の i
int upper_bound(int *a, int sz, int x) {
  int m, l = 0, r = sz;
  while (l+1 < r) {
    m = (l+r)/2;
    if (a[m] <= x) l = m; else r = m;
  }
  return a[l] <= x? r: l;
}
```

上記の C 言語による関数の記述では、引数はそれぞれ、データ配列 a[]、データ配列の個数 sz、探索キー x を意味する。

探索キーが最小値を下回り、あるいは、最大値を超えた場合の振る舞いが表 1 のとおり、網掛けの値になる。

なお、ここでは割愛するが、a[index]<=key という条件を満たす最大の index を求める関数も、a[index]<key という条件を満たす最大の index

を求める関数も上記の関数を若干修正することで実現できる。

4 ソートアルゴリズム

n 番目だけでなく、全データを整列させるのがソートアルゴリズムである。最もよく知られているのはクイックソートというアルゴリズムであり、時間計算量は平均 $O(n \log n)$（n はデータの個数）である。より高速なソートアルゴリズムは他に存在するが、汎用性ではクイックソート以上のものはまだ現れていない。

例として、最高速のソートアルゴリズムのひとつである分布数えソート（計数ソートともいう、counting sort）を紹介する。時間計算量は最良でも最悪でも $O(n+k)$ 程度である。ただし、k はデータの値の分布範囲（データの最大値−最小値の値）を示す。

分布数えソートは利用できる前提条件がやや厳しい。つまり、データの値の分布が小範囲に留まることが要求される。データの個数 n をデータの値の分布範囲 k が大幅に超えたら、分布数えソートの高速性が失われる。たとえば、データ配列 a[]={0, 2^{32}} に対しては、分布数えソートを使うべきではない。

```
// 分布数えソート O(n+k)
int f[MAXV];
void counting_sort(int *ans, int *a,
int sz, int vmin, int vmax) {
  int i, t;
  for (i=0; i<=vmax-vmin; i++) f[i]=0;
  for (i=0; i<sz; i++) f[a[i]-vmin]++;
  for (i = 1; i <= vmax-vmin; i++)
     f[i] += f[i-1];
   for (i = sz-1; i >= 0; i--) {
   t = a[i]-vmin;
   ans[--f[t]] = t;
  }
}
```

上記の C 言語による関数の記述では、引数はそれぞれ、ソート済データ配列 ans[]、元データ配列 a[]、データ配列の個数 sz、データの最小値 vmin、データの最大値 vmax を意味する。作業用配列 f[MAXV] はデータ配列 a の値（最大値−最小値）以上の大きさを確保すべきである。関数の内部に目を向けて確認すると、4 つの for 文のうち、2 つはループ回数が（最大値−最小値）程度であり、残りの 2 つはデータの個数である。したがって、全体の計算時間量が $O(n+k)$ となる。

データ範囲の制限を緩和するソートアルゴリズムとして、基数ソート（radix sort）というものが提案されている。データの値を下の桁から順番にソートしてゆき、最後に最上位桁でソートすると、全体が順序通りに並ぶ、という手法である。たとえば、符号無し 64 ビット整数は 20 桁の 10 進数に相当するので、1 桁ずつ順番にソートしていくと、全体の計算時間量は桁数の分しか増加しないため、$O(kn)$（k は桁数）になる。

5 二分探索木

二分探索木（binary search tree）は、ひとつのノードから、左の子孫となるノードと、右の子孫となるノードの 2 つしかなく、しかも、左の子孫の値 ≤ 親の値 < 右の子孫の値という制約をもつ木の構造であり、探索木のうちで最も基本的なものでもある。

左右の部分木がバランスをとれている状態では木全体の高さは $O(\log_2 n)$ であるが、最悪の場合、たとえばそれぞれのノードにおいて、左の部分木しか存在しないという状況になると、木の高さは $O(kn)$ となる。木の形はノード挿入時のデータ値の出現順序に依存するため、挿入・削除の際に木のバランスを取り直す処理を追加する必要がある。

バランスの取れた二分探索木において、n 番目に小さい値は計算時間量 $O(\log n)$ で求めることができる。GNU GCC では find_by_order という関数がそれに相当する。

6 BIT 木（Binary Indexed Tree）

BIT 木（Binary Indexed Tree）、またはフェニック木（ Fenwick Tree）とは、部分和の計算と要素の更新の両方を効率的に行える木構造である。ピーター・フェニックにより 1994 年に提案された（Fenwick 1994）。つまり、データ配列 a[i] に対して、以下のクエリを計算時間量 $O(\log n)$ で実現するものである。

sum（i）：$\sum_{j=1}^{i} a_j$ を求める（累積和）
add（i,x）： a[i] に x を加える

C 言語の配列は一般的にインデックス 0 からデータを格納していくが、BIT 木ではインデックス 1 からデータを格納することに留意しよう。また、デー

タ配列の累積和は $O(\log n)$ で求まるということを示したということでもない。BIT 木を構築するのに、それぞれのデータを対し、$O(\log n)$ の計算が必要で、すべてのデータを BIT 木に格納するのに、$O(n \log n)$ の計算時間量が必要であるからである。データ配列の累積和だけを求めるなら、最速で自明な $O(n)$ である単純な加算の計算には敵わない。

BIT 木は威力を発揮するのは、BIT 木の構築が完了後に、`add(i,x)` と `sum(i)` のクエリそれぞれに、$O(\log n)$ で回答できることである。従来の方法での、`add(i,x)` が $O(1)$、`sum(i)` が $O(n)$ であることと対比してその高速性が理解されよう。

BIT 木の仕組みは図 1 のように、2 番目のデータが 1 番目のデータとの部分和、4 番目のデータは 1 番目～ 4 番目の部分和、8 番目のデータは 1 ～ 8 番目の部分和となり、2 進数の特性をうまく利用して、累積和の高速計算を実現したのである。

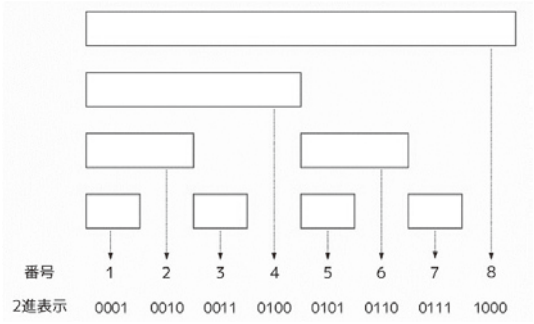

図 1　Bit 木の仕組み

C 言語による BIT 木の実装はつぎのとおりである。

```
// Bit木 library
int bit[MAXLEN], sz;
int sum(int i) {
  int s = 0;
  while (i > 0)
    s += bit[i], i -= i & -i;
  return s;
}
void add(int i, int x) {
  while (i <= sz)
    bit[i] += x, i += i & -i;
}
```

`MAXLEN` は配列の長さ以上に指定する定数であり、変数 `sz` はデータ配列の実際の長さに一致させる。`bit` 配列は図 1 に対応していて、初期状態では各要素の値を 0 にすべきである。

BIT 木の高速性を活用した、n 番目に小さい値を求めるアルゴリズムをつぎに示す。

```
#define MAXVAL 10000005
int bit[MAXVAL], sz, p;
void init(int maxVal) {
  sz = maxVal;
  p = 1; while (p < sz) p <<= 1;
}
void add(int i, int x) {
  i++; while (i <= sz)
    bit[i] += x, i += i & -i;
}
void insert(int val) { add(val, 1); }
void erase(int val) { add(val, -1); }
int nth_element(int n) {
  int a = 0, q = p; n--;
  while (q >>= 1)
    if (a+q <= sz && bit[a+q] <= n)
      a += q, n -= bit[a];
  return a;
}
```

データの最大値が 1000 万以下の入力に対処できるよう、MAXVAL を指定したが、使用状況に応じて適宜修正してよい。

さて、BIT 木による n 番目に小さい値を求めるアルゴリズムの使い方は次のとおりである。

0. データの最大値で初期化：`init(maxVal)`。
1. 要素（非負値 `val`）を追加する操作：`insert(val)`。
2. 要素（非負値 `val`）を削除する操作：`erase(val)`。
3. n 番目に小さい値を取得する操作：`nth_element(n)`。

要素の値が重複しても問題ない。つまり、BIT 木が扱っているのは集合ではなく、多重集合（マルチ集合、要素の重複を許す集合）と見なしてよい。

そして、各操作の計算時間量は次に示すとおり、どれも $O(\log n)$ である。ただし、n はデータの最大値を表す。

追加操作 `insert(val)` は $O(\log n)$。

削除操作 `erase(val)` は $O(\log n)$。

取得操作 `nth_element(n)` は $O(\log n)$。

値の分布範囲に比べて、データの個数が少ない状況では、ハッシュテーブル（Cormen 2010）にマッピングさせる等の技法が有用になるかもしれない。

実行例として、初期化後、6つのデータ 5, 2, 0, 10, 2, 6 を追加し、*n* 番目に小さい値を以下のテストプログラムで求めてみた。実行結果となる画面表示は 0, 2, 2, 5, 6, 10 であった。

```
//  テストプログラム
int data[6] = {5,2,0,10,2,6};
int main() {
  int i;
  init(10);   // 入力データの最大値以上
  for (i = 0; i < 6; i++)
   insert(data[i]);
  for (i = 1; i <= 6; i++)
   printf("%d ", nth_element(i));
  printf("\n"); return 0;
}
```

選択アルゴリズムに比べ、BIT 木の優位性は高速性だけでなく、削除操作に対応することも注目に値する。なぜなら、選択アルゴリズムにおいて、データ配列から特定の要素を削除するクエリを考慮しておらず、リニアサーチでは特定の値を調べるには、データ配列に格納される要素をすべてアクセスするので、平均でも最悪でも計算時間量は $O(n)$ になってしまう。それに、データ配列が整列されていない状態ではバイナリサーチは使えない。

従って、要素の追加と削除に関するクエリが大量に繰り返し現れる状況では、選択アルゴリズムに比べ、BIT 木による方法が圧倒的に高速である。そのことは以下Ⅲの比較実験で検証する。

また、二分探索木に比べ、計算時間量は同一オーダであるが、実際には二分探索木にバランスの取り直す処理や実装の複雑性等の問題が纏わる。

Ⅲ 比較実験

ここでは、選択アルゴリズム、二分探索木、BIT 木によるアルゴリズムの実行時間について比較実験を行う。二分探索木は実装の仕方によって性能が大きく異なるので、GNU GCC のライブラリをそのまま使用することにした。アルゴリズムの計算結果を相互に比較することで計算間違いを防ぐ。

1 昇順線形データ（追加＋取得操作）

0 から 100 万までの整数をひとつずつ追加する。値を 1 つ追加した後、最小値（n = 1）、最大値（n = N）、中央値（n = N/2）を求める。ただし、N はその時点で追加したデータの総数である。上記の操作を 10 回繰り返し、その計算時間の平均値をそれぞれのアルゴリズムの実行時間とする。実験結果は表 2 に示す。

表 2 昇順線形データに対する比較実験

アルゴリズム種別	平均計算時間（秒）
選択	3204.16
二分探索木	5.41
BIT 木	0.42

PC 環境によって計算時間が大きく変わるので、実験結果はあくまでもひとつの目安に過ぎない。しかし、あまりにも選択アルゴリズムと BIT 木アルゴリズムとの時間差が大きかった。計算時間量 $O(n)$ と $O(\log n)$ との差がはっきりと表れていることといえよう。

2 降順線形データ（追加＋取得操作）

上記の 1 と同様の実験だが、データは 100 万から 0 までの整数に、つまり、昇順データを降順に変えて実験を行い、表 3 を得た。

表 3 降順線形データに対する比較実験

アルゴリズム種別	平均計算時間（秒）
選択	3313.68
二分探索木	4.63
BIT 木	0.49

BIT 木が表 2 より時間がかかったのは、毎回最大値からの取得になるからであろう。表 3 でも選択アルゴリズムは実行時間があまりにもかかることがわかったので、以下の実験ではそれを外すことにした。

3 ランダムデータ（追加＋取得操作）

上記の 1 と同じ条件での実験であるが、追加するデータは線形データではなく、下記の疑似乱数生成アルゴリズム *xorshift* によるランダムデータを使用する（ただし、最大値を超えないよう、生成した乱数 y に対し、y % 1000001（% は剰余演算子）を使用した）（倪 2018）。実験結果は表 4 に示す。

```
unsigned int xorshift() {
```

```
  static unsigned int y = 2463534242;
  y = y ^ (y << 13);
  y = y ^ (y >> 17);
  y = y ^ (y << 5);
  return y % 1000001;  // 最大値の制限
}
```

表 4 ランダムデータに対する比較実験

アルゴリズム種別	平均計算時間（秒）
二分探索木	3.98
BIT 木	0.39

実験でわかったことは、重複データの追加は GNU GCC のライブラリーでは許されていないことである。対して、BIT 木では重複データでも正常に機能した。

4 ランダムデータ（追加＋削除＋取得操作）

1 から 100 万個までのランダム整数をひとつずつ追加する。その後、100 万から 200 万個までのランダム整数を 1 つずつ追加した直後に、中央値を取得してそれを削除して、最小値（n = 1）、最大値（n = N）、中央値（n = N/2）を求める。ただし、N はその時点のデータの総数である（後半ではランダム数をひとつ追加して、ひとつ削除するので、N の値は 100 万のままである）。上記の操作を 10 回繰り返し、その計算時間の平均値をそれぞれのアルゴリズムの実行時間とする。実験結果は表 5 に示す。

表 5 ランダムデータに対する追加削除取得実験

アルゴリズム種別	平均計算時間（秒）
二分探索木	7.59
BIT 木	0.69

表 4 よりデータの数が倍になったので、実行時間も倍近くなった。

5 実験データから得られたこと

上記の実験でわかったことをまとめて列挙する。

1. 選択アルゴリズムは計算時間量が $O(n)$ ということで、他の 2 つのアルゴリズムに比べて処理時間がとてもかかった。
2. GNU GCC で実装された 2 分探査木は重複データの入力に対応していない。重複データに対応するには何らかの対策が必要である。
3. 本文の検討対象である n 番目に小さい値を求めることに関し、BIT キーによるアルゴリズムは速さ、扱いやすさの面において最も優れている。ただし、データの個数に対して、データの値の分布範囲が広い場合に、ハッシュテーブルを使うなり、工夫を必要とする場面もあろう。逆に、データの個数に比べて、データ値の分布範囲が狭い場合、さらなる高速性が発揮されうる。

終わりに

最小値、最大値、中央値等、n 番目に小さい値を知りたい場面は決して少なくない。本文では順序づけに関連する関数やアルゴリズムを考察し、大量のデータが個々に追加、削除され、それぞれの操作の直後に n 番目に小さい値を効率的に求め、時間計算量が $O(\log n)$ である BIT 木によるアルゴリズムを説明した。その高速性や使いやすさ、実装のしやすさ等が 2 分探索木よりも優れていることを比較実験でも確かめることができた。多くの場面で活用されることを期待する。

参考文献

秋葉拓哉、岩田陽一、北川宜稔（2010）、『プログラミングコンテストチャレンジブック（第 2 版）』マイナビ出版

奥村晴彦（1991）、『C 言語によるアルゴリズム事典』技術評論社

倪永茂（2018）、「C 言語プログラミング実践教育におけるビット演算」宇都宮大学国際学部研究論集 46, 87-94.

Comen, Thomas H., Leiserson, Charles E., Rivest, Ronald L. and Stein, Clifford (2001), *Introduction to Algorithms, Second edition*, The MIT Press.

Fenwick, Peter M. (1994), A New Data Structure for Cumulative Frequency Tables, *Software: Practice and Experience*, Vol. 24 (3), 327-336.

Laaksonen, Antti (2018), *Guide to Competitive Programming: Learning and Improving Algorithms Through Contests*, Springer.

Algorithms for Finding nth Smallest Element in Unsorted Array

NI Yongmao

Abstract

In this paper, we focus on 3 different types of algorithms to find n'th smallest element in unsorted array, 1) selection algorithm (time complexity *O(n)*), 2) binary search tree *O(log n)*, and 3) the algorithm based on Binary Indexed Tree (BIT, or Fenwick Tree) *O(log n)*. The BIT algorithm is suitable for a large amount of queries of add, remove or update operations with complexity *O(log n)*. And we present the results of the practical efficiency of these algorithms.

（2019 年 5 月 27 日受理）